

Devomorph User's Guide

Patrick Jayet and Ronney Meier

October 2005

1 Introduction

Devomorph is a framework which uses Genetic Algorithms (Artificial Evolution), a GRN (Genetic Regulatory Network) and a physics engine to evolve agents, which interact with their environment and perform certain tasks. These agents could even have the possibility to change their shape while interacting with the environment.

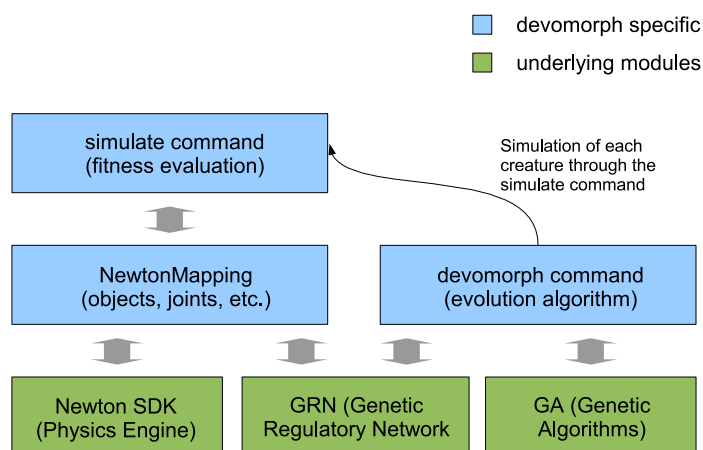


Figure 1: The overall scheme of the devomorph project. Two command line tools, `devomorph` and `simulate` represent the project entry points.

2 NewtonMapping Classes

The `NewtonMapping` classes aim at providing an abstraction layer to the `Newton SDK` – which is the physics engine used to perform the simulation of each artificial being –, to the genetic algorithms in broad sense, used to encode and express the properties of each being (through a genetic regulatory network or GRN) and to perform the evolution steps of the population (through a specific genetic algorithm or GA).

The following classes are part of the `NewtonMapping` namespace. A short explanation is given to illustrate the role of each one. The classes are separated between generic ones, that implement common generic functionalities, and specific ones, that extend the generic ones and are specific to a model.

Generic classes:

- **ModelAdapter**: the `Newton` engine works through callbacks, which are a lot easier to realize in C++ if the callback methods are static. The `ModelAdapter` has the task to let the non-static callback methods of classes like `Model` be called through its static methods. This adapter class contains internally an instance of a `Model`.

- **Model**: abstract class that implements all methods needed by Newton that are not specific to a given model and defines the other methods as abstract and virtual such that a specific model, derived from **Model**, has to implement them.
- **Object3D**: abstract class for a building block of an artificial being.
- **Joint**: abstract class for a joint between two or more building blocks, dependant on the model.
- **RotationMatrix**: extension of Newton's **dMatrix** class. It is used to rotate simple vectors or even complete matrices around an arbitrary axis in space. This class is heavily needed to orient new objects or to set the line of action (screw) of joints in space.
- **ProteinMapping**: implements the mapping between the protein indices in the model and the indices in the GRN. For using another mapping this class needs to be derivated.

Specific classes

- **TriangleModel**: class extending **Model** and implementing some virtual abstract methods from **Model**. An example of such a method is `getFitness()` that calculate the fitness value of a being. This information is specific to a certain model and is therefore implemented in **TriangleModel**.
- **Prism**: class extending **Object3D** and implementing the desired functionalities of a prism object
- **HingeJoint**: class extending **Joint** and implementing the functionalities specific to a hinge joint between two prisms. Could also be used for other extensions of **Object3D**.

Illustration 2 present a complete UML scheme of the classes from **NewtonMapping**.

3 Programming a new Model for **NewtonMapping**

For creating a new model a good point to start is to look at the examples **TriangleModel**, **HingeJoint** and **Prism**. In these files are also a lot of examples of how to implement certain functionalities in Newton. One should also look at the first few simple tutorials of Newton to get a feeling of how Newton works. The Newton documentation is in **NewtonSDK/doc** and the examples are in **NewtonSDK/samples**.

For programming a new model, one propably will need the model itself, derivated from **Model**, as well as a new object-model and joint-model, derivated from the **Object3D** and **Joint** basis classes. There is also the possibility to use one of the allready created object or joint classes – details about the model that we use presently, the triangle model, can be found in section 5. The derivated class from the **Model** class is responsible for creating, changing and controlling the whole agent. It uses the derivated joint and object classes to generate the agent and to get information about their state. In the derivated class all the abstract methods (signalized by the **virtual** and **= 0** keywords) needs to be implemented and the virtual methods (with just the **virtual** keyword) do not need to be re-implemented.

The most important methods of the **Model** class (for other methods and properties take a look at **Model.h**):

- **virtual void DrawScene() = 0**: This is the method which will be called for each step of the simulation. It contains the whole code for drawing the scene and also for controlling/changing the agent. The code for drawing, saving pictures for creating a movie and other features can be copied out of the implementation in **TriangleModel.cpp**.
- **virtual unsigned JointUserCallback (...) = 0**: This callback is called by Newton on each simulation step for each joint for which this callback is set as `usercallback` at creation time. It can be used to control the constraint of the joint. Certain functioncalls which affects the behaviour of a joint can be called only from here.

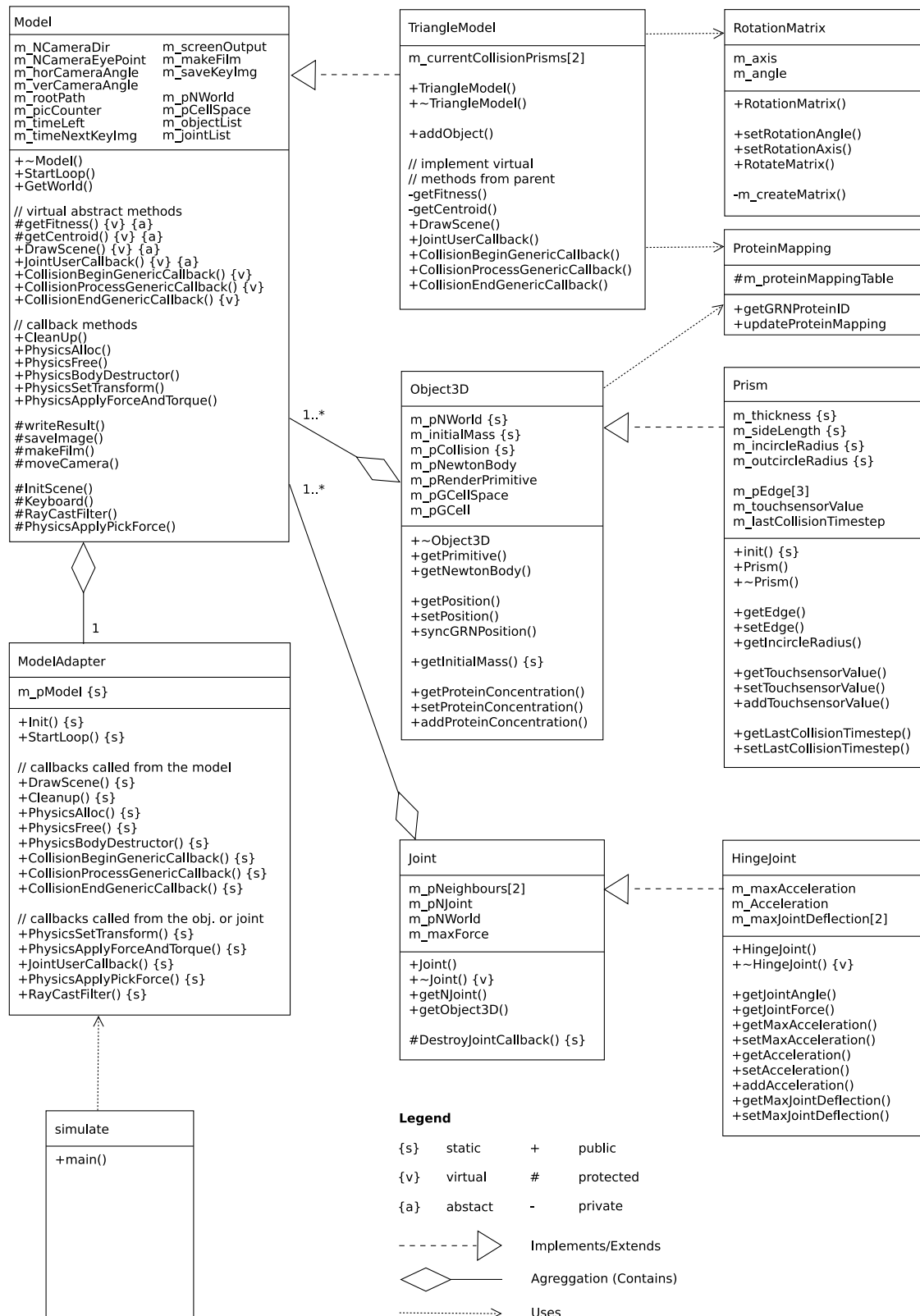


Figure 2: UML scheme of the classes from the NewtonMapping namespace

- `virtual int CollisionBeginGenericCallback (...)`: This is a callback which Newton will call on each simulation step for each collision. It can be set only for whole materialgroups, not for single objects. In devomorph it is set for the default material, which is used for objects if they do not get an other material assigned. So for creating objects which should use an other callback or none at all, one need to use another material. More information are in the Newton help about `NewtonMaterialSetCollisionCallback`.

This callback will be called before the contact calculation of the collision. If the application must process the contacts after they have been calculated, 1 is to be returned. Otherwise newton won't call `CollisionProcessGenericCallback` for this object. In `TriangleModel` the generic collision callbacks are used for implementing a touch sensor. Some information and functionalities are only available in these callbacks.

- `virtual int CollisionProcessGenericCallback (...)`: This callback is called by Newton in each step for each contact between two bodies. Return 1 to tell Newton that we want to accept this contact. This callback does not need to be implemented.
- `virtual void CollisionEndGenericCallback (...)`: Newton will call this callback after all contacts of a collision have been processed. This callback does not need to be implemented.
- `virtual ~Model()`: This is the deconstructor of the Model. **This is not clear, the deconstructor of the model subclass should free the resources of the subclass (has to be implemented), but nothing needs to be done for the Model, whose destructor just frees the resource of the Model itself. At the end, both destructor are called, 1st the one of the subclass, then the one of the parent** It should be implement for cleaning up the memory after the simulation of an agent ends. This method does not need to be implemented.

For writing own object and joint classes, one can just derivate them from the `Object` and `Joint` basisclass. The whole functionality for communicating with the GRN is already implemented there.

4 The Devomorph Command

The file `devomorph.cpp` is the main entry point of the Devomorph project, implementing the main loop of the genetic algorithm. At the beginning, a new population is initialized with a random genetic code, then N generation loop are performed in order to develop, select, reproduce and mutate the population. The develop step is then performed by calling the `simulate` command, that makes a 3D simulation of the specified artificial being, and by reading the corresponding fitness value from the simulation. The parameters that the command `devomorph` accepts are the following:

Start a new simulation:

```
devomorph new <nb of generations> <nb of beings> <time to run> [options]
```

Resume a simulation:

```
devomorph resume <gen to resume> <nb of generations> <time to run> [options]
```

DESCRIPTION

<nb of generations>: number of generations to run for the genetic algorithm

<nb of beings>: number of beings for each generation

<gen to resume>: in case of a resume, specify the root folder of the generation to resume from, i.e. `Result20050921-15h28/G110`

<time to run>: time in seconds for each simulation to run (e.g. 30)

OPTIONS

<code>--help</code>	show this usage
<code>--makemovie</code>	save the images during the simulation and generate a movie at the end (may take some time), default: no rem: it would be a better idea to let the set of simulations run without this option and to make a movie of the best beings using directly the simulate command with the <code>--makemovie</code> option
<code>--takeimage</code>	save key images of the scene at regular intervals, default: no
<code>--noscreen</code>	the simulation produces no drawing output (useful when running in batch mode), default: no

5 The Simulate Command

The `simulate.cpp` command is the main entry point of the simulation. A new Model is created via the adapter class `ModelAdapter` and the Newton 3D engine started. After the given amount of time is elapsed, the `TriangleModel` class calculates the fitness value of the simulated artificial being and write it in a file, in the root folder of that being. Every input/output of `simulate` is done through reading and writing files, as this is some kind of easy inter-process communication that is not dependent of a platform, as the Devomorph project can be used under the Windows or Linux operating system. The genetic code that `simulate` needs as input is also read from a file in the root folder of that being.

The parameters that `simulate` accepts are the following:

Usage: `simulate <root path> <max time to run> [options]`

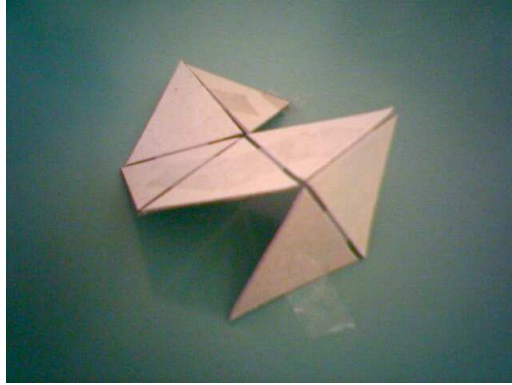
OPTIONS

<code>--help</code>	show this usage
<code>--makemovie</code>	save the images during the simulation and generate a movie at the end (may take some time), default: no
<code>--moviehi</code>	if the option <code>--makemovie</code> has been specified, the size of the movie created is 640x480 at the place of the default 320x240 (may take a lot of time)
<code>--takeimage</code>	save key images of the scene at regular intervals, default: no
<code>--noscreen</code>	the simulation produces no drawing output (useful when running in batch mode), default: no

6 The Triangle Model

6.1 Model

The triangle model has been written as a model which could be implemented in hardware. It doesn't use a neural network but only the GRN, which not only controls the shape of the agent but also controls its movement.



The only task the agent has, is to move as far as possible. The single body parts are prisms with a triangular base and hingejoints as connectors. The number of possible body parts is bounded to 10 (constant `MAX_BODY_PARTS` that is defined in `TriangleModel.cpp`) but could be changed if needed. In the following table, each protein index and the effect of the corresponding protein is described.

Protein	Effect	Implementation Details
0	attach new triangle to edge A if concentration is over a certain threshold (only if there isn't any connection already at the respective edge)	before the triangle is created it is calculated if it would be created in the floor. If that is the case, the triangle is not created at this moment
1	attach new triangle to edge B if concentration is over a certain threshold (only if there isn't any connection already at the respective edge)	idem
2	attach new triangle to edge C if concentration is over a certain threshold (only if there isn't any connection already at the respective edge)	idem
3	product of the concentrations in two interconnected triangles determines force into one direction	
4	product of the concentrations in two interconnected triangles determines force into the other direction	
5	product of the concentrations in two interconnected triangles determines the spring constant of the joint on one side	when calculating forces with the spring constant we do not use distances but the joint deflection in radians
6	spring constant on other side	idem
7	concentration determined or at least influenced by joint stress	
8	concentration influenced by touch sensor activation	the touch sensor calculates the total pressure on a prism

6.2 Drawbacks

- **Explosion of the physics engine** At the beginning we had a protein which was responsible for setting the maximal deflection of a joint. In Newton this constraints are hold by using a certain acceleration as soon as a joint is over it's constraint. This behaviour of Newton connected with the ability of the agent to change this constraints can lead to an unstable

simulation. If for example a Joint was already at it's maximal deflection and the agent decided to change the maximal deflection to a value, that is a lot smaller, Newton needs to use a huge acceleration to get the Joint back into it's constraint. This huge acceleration can lead to an explosion of the system.

- **Choice of a good fitness function** The actual fitness function, the distance between the centroid of the creature at the end of the simulation and the initial position of the first triangle can lead to an agent becoming an acceptable fitness value just by growing only on one side, behavior which we are not interested in. A slightly more complicated fitness function, like the mean distance of each prism between its start and end position would be a better candidate.

References

- [BP01] Bongard, J. and Pfeifer R., *Evolving Complete Agents using Artificial Ontogeny*, First International Workshop on Morpho-Functional Machines, Tokyo, Japan, 2001
- [BP] Bongard, J. and Pfeifer R., *Environmental Shaping during Artificial Ontogeny*, AILab, University of Zurich
- [LEC04] Leclerc, R., *Extending Evolvability in Artificial Embryogeny for Artificial Life Models*, Master Thesis, February 2004
- [SIM94A] Sims, K., *Evolving Virtual Creatures*, Computer Graphics, Annual Conference Series, July 1994, pp.15-22
- [SIM94B] Sims, K., *Evolving 3D Morphology and Behavior by Competition*, Artificial Life IV Proceedings, 1994, pp.28-39